Efficient Schemes for Computing α-tree Representations

Jiří Havel, François Merciol, Sébastien Lefèvre

Efficient Schemes for Computing α-tree Representations

Jiří Havel[1], François Merciol[2], and Sébastien Lefèvre[2]

[1] Brno University of Technology, Czech Republic
ihavel@fit.vutbr.cz
[2] Université de Bretagne-Sud, IRISA, France
{francois.merciol,sebastien.lefevre}@univ-ubs.fr

Abstract. Hierarchical image representations have been addressed by various models by the past, the max-tree being probably its best representative within the scope of Mathematical Morphology. However, the max-tree model requires to impose an ordering relation between pixels, from the lowest values (root) to the highest (leaves). Recently, the α-tree model has been introduced to avoid such an ordering. Indeed, it relies on image quasi-flat zones, and as such focuses on local dissimilarities. It has led to successful attempts in remote sensing and video segmentation. In this paper, we deal with the problem of α-tree computation, and propose several efficient schemes which help to ensure real-time (or near-real time) morphological image processing.
Key words: α-tree; Quasi-Flat Zones; Image Partition; Hierarchies; Efficient Algorithms.

1 Introduction

Mathematical morphology has long been a provider of interesting hierarchical image representations, mainly by trees, e.g. component tree [4], min and max-tree [9], binary partition tree [8], etc. The max-tree (and its respective counterpart, min-tree) has been widely used due to its nice properties as well as the availability of efficient algorithms to first compute the tree from an image, and second process the tree (e.g. with a filtering to remove irrelevant nodes), thus leading to the processing of the underlying image.

Recently, a new image model, namely the α-tree [6], has been introduced to avoid relying on an ordering relation among image pixels. This model is a hierarchical representation of the quasi-flat zones of an image, and as such, relies on local dissimilarities α. While this model already led to successful attempts in exploration of remote sensing data [5] and image/video segmentation [3], it still requires some efficient computing schemes in order to be widely disseminated and to address a large scope of problems. In this paper, we focus on computational issues of the α-tree model, and propose new efficient algorithms to build this α-tree.

The rest of this paper is organized as follows. Section 2 provides necessary background on quasi-flat zones and α-tree. Various schemes for computing efficiently the α-tree are discussed in Section 3. Section 4 is devoted to the presentation of experimental results, while concluding remarks are given in Section 5.

2 Background

The α-tree image model is a multiscale representation of an image through its α-zones. We recall here the notions of flat zones, quasi-flat zones (including α-zones) and finally the recent α-tree model.

In the following, we will use the notations used in [6]. We will denote by I a digital image and E its definition domain. Let us recall that an image segmentation is a partition $\mathbf{P}$ of E, i.e. a mapping $x \rightarrow \mathbf{P}(x)$ from E into $\mathcal{P}(E)$ such that $\forall x \in E \Rightarrow x \in \mathbf{P}(x)$ and $\forall x, y \in E \Rightarrow \mathbf{P}(x) = \mathbf{P}(y)$ or $\mathbf{P}(x) \cap \mathbf{P}(y) = \emptyset$, with $\mathbf{P}(x)$ indicating a cell of $\mathbf{P}$ containing a point $x \in E$. We thus have $\bigcup_{x \in E} \mathbf{P}(x) = E$. Moreover, we will write $\pi(x \rightsquigarrow y)$ a path of length N between any two elements $x, y \in E$, i.e. a chain of pairwise adjacent elements $\langle x = x_0, x_1, \ldots, x_{N-1} = y \rangle$. Finally, let $\mathit{\Pi} \neq \emptyset$ be the set of all possible paths between x and y. The minimum dissimilarity metric between x and y is defined as

$$\hat{d}(x, y) = \bigwedge_{\pi \in \mathit{\Pi}} \left\{ \bigvee_{i \in [0, \ldots, N-1]} \left\{ d(x_i, x_{i+1}) \mid x_i, x_{i+1} \in \mathit{\Pi} \right\} \right\} \tag{1}$$

with $d(x, y)$ a predefined dissimilarity measure between attributes of x and y (i.e. pixel intensities).

In a digital image, flat zones are defined as connected sets of pixels sharing the same value. Formally, the flat zone of x is defined as

$$\mathcal{Z}(x) = \{x\} \cup \{y \mid \exists \pi(x \rightsquigarrow y) : \forall x_i \in \pi(x \rightsquigarrow y) \wedge x_i \neq y \Rightarrow d(x_i, x_{i+1}) = 0\}. \tag{2}$$

In the field of Mathematical Morphology, flat zones have been shown to be elements with nice properties [10]. Indeed, the partition of an image into its flat zones most often includes any relevant image segmentation, since objects edges are located between neighboring pixels with different values, i.e. belonging to different flat zones. However, the practical usage of flat zones is limited since it leads to an extreme oversegmentation, flat zones being made of only a few pixels. To counter this problem, softer definitions have been introduced under the name quasi-flat zones. A recent survey related to quasi-flat zones is provided by Soille in [11].

The simplest and most widely used definition of quasi-flat zones is called α-zone. For a given pixel x, its α-zone noted α-$\mathcal{Z}(x)$ is made of all pixels reachable from x through a path with intermediary steps not higher than α. Using the previous definitions, we have

$$\alpha\text{-}\mathcal{Z}(x) = \{x\} \cup \{y \mid \exists \pi(x \rightsquigarrow y) : \forall x_i \in \pi(x \rightsquigarrow y) \wedge x_i \neq y \Rightarrow d(x_i, x_{i+1}) \leq \alpha\}, \tag{3}$$

the specific case of $\alpha = 0$ leading to standard flat zones. Let us observe that α-zones define a partition or segmentation, i.e. $\bigcup_{x \in E} \alpha\text{-}\mathcal{Z}(x) = E$ and $\forall x, y \in E$: $\alpha\text{-}\mathcal{Z}(x) \cap \alpha\text{-}\mathcal{Z}(y) \neq \emptyset \implies \alpha\text{-}\mathcal{Z}(x) = \alpha\text{-}\mathcal{Z}(y)$. The main drawback of α-zones is

their purely local behavior, which can lead to an artifact called chaining effect. This is observed when the successive steps in a path $\pi(x \rightsquigarrow y)$ are low (w.r.t α) while the dissimilarity measure between x and y is high, for instance in the case of a gradual transition from black to white. This limitation was addressed by Soille [11], and later Soille and Grazzini [12], who added another constraint that prevented the uncontrolled α-zone from growing.

Another way to use the α-zones has been recently reported by Ouzounis and Soille in [6]. In this seminal work, they introduce the concept of α-tree based on a partition pyramid. Since the α zones can be ordered by inclusion relation, it is possible to construct a tree of α-zones. The root of the tree is a zone covering the whole image. Every parent node contains a zone, that is a superset of zones contained in its children.

Every level of the α-tree contains all zones for a specific value of α. The α-tree contains all possible image segmentations based on α-zones. Every cut through the α-tree selects an image segmentation. To benefit from this powerful image representation, efficient computation schemes are required.

3 Computing the α-tree

3.1 Basic Principles

In the previous section, we have introduce the α-tree reusing its standard notations [6]. Let us observe that the α-tree can also be defined through graphs. We will use this latter representation to introduce efficient computation scheme. Thus, let us also denote an image by a graph $(\mathcal{V}, \mathcal{E}, \mathcal{I})$, where $\mathcal{V}$ are the image pixels, $\mathcal{E}$ are the edges between them and $\mathcal{I} \subseteq \mathcal{V} \times \mathcal{E}$ is the incidence relation between vertices and edges, i.e. for edge e, that connects v_1 and v_2, $v_1 \mathcal{I} e$ and $v_2 \mathcal{I} e$. The α is a weight of the edges of the graph.

The α-tree can be constructed as a min-tree built over an edge graph, as outlined by Soille and Najman in [13]. An edge graph for a graph $(\mathcal{V}, \mathcal{E}, \mathcal{I})$ is a graph $(\mathcal{V}', \mathcal{E}', \mathcal{I}')$, with edges and vertices exchanged $(\mathcal{V}' = \mathcal{E}, \mathcal{E}' = \mathcal{V})$, while the incidence is preserved $(e \mathcal{I}' v = v \mathcal{I} e)$. Using this definition, the edge graph is actually a multigraph, i.e., for a 4-connected image the edges connect up to four vertices. The min-tree of the edge graph can be then transformed in an α-tree. As Soille and Najman mention in [13], this indirect construction is unnecessary, but their paper does not provide a direct algorithm.

It is possible to build the α-tree directly by a modification of Tarjan's union-find algorithm [14]. Tarjan's algorithm was initially defined to identify connected components, it has been later used for construction of different component trees, e.g. in [4]. The union-find method processes a list of graph edges sorted by increasing alpha. For each edge it finds two largest connected components, the edge connects, and merges them. Since the algorithm works with sorted edges, the α-tree is built bottom-up from fine to coarse subdivision of the image.

An edge of the image graph connects two neighboring pixels. Algorithm 1 shows the basic α-tree construction. The parameters of an edge are its two endpoints and the corresponding α. The tree is built bottom-up by subsequent merges of partial trees. For every edge, the two partial trees that contain the edge endpoints are found and then merged. When a pixel is not yet linked to a leaf node, the node is created and linked

from the leaf array. Due to the ordered edges, the nodes will be merged only in the active layer. It is the layer of the tree with α being equal to the α of the processed edge. The levels of the tree below the active level are never changed again.

To ensure, that each path from a root to a leaf contains at most one node per α-level, there are four possibilities of merging the subtrees. If both tree roots have α less than the processed edge, a new node with both trees as children is created. If only one tree root has α less than the processed edge, then this tree is attached as a child to the other tree root. Finally if both roots have α equal to the processed edges, then the two root nodes are merged (i.e. their children attached to one of them and the other marked for removal)

Algorithm 1 Algorithm for α-tree construction

Require: Image I
Ensure: Array leaves[height(I), width(I)] of Node pointers
  leaves$[i, j] :=$ null, $\forall i, j$
  $E = \text{edges}(I)$
  sort E by ascending α
  for $e = e_1 \rightarrow e_{|E|}$ **do**
    $p_1, p_2 = \text{points}(e)$
    $n_{\{1,2\}} = \text{findRoot}(p_{\{1,2\}})$
    if $n_i = \text{null}$ **then**
      leaves$[p_i] = n_i = \text{makeNode}(p_i, \alpha(e))$
    end if
    if $n_1 \neq n_2$ **then**
      if $\alpha(n_1) = \alpha(n_2) = \alpha(e)$ **then**
        $\text{merge}(n_1, n_2)$
      else if $\alpha(n_1) < \alpha(e)$ and $\alpha(n_2) < \alpha(e)$ **then**
        $\text{makeNode}(n_1, n_2, \alpha(e))$
      else if $\alpha(n_1) = \alpha(e)$ and $\alpha(n_2) < \alpha(e)$ **then**
        attach n_2 to n_1
      else if $\alpha(n_1) < \alpha(e)$ and $\alpha(n_2) = \alpha(e)$ **then**
        attach n_1 to n_2
      end if
    end if
    Update path compression cache
  end for

When the α is a small integer, the edges can be ordered in $O(E)$ time using the bucketsort algorithm. This algorithm first calculates a histogram of edge alphas. Calculating the prefix sum converts the histogram to a list of offsets to an output array. The reordering is then done in the following pass.

Finding the root of a subtree for a given pixel contains potentially lengthy bottom-up tree traversal. The traversal can be optimized by path compression. During the tree construction, a cache exists, that links tree nodes to the subtree roots. After each processed edge, this cache is updated. Because the number of affected pixels is large and a full update would be costly, only portion of the affected nodes is updated. Thus, the

cache does not remove the bottom-up tree traversal, but can significantly shorten it. On the other hand, nodes can't be deleted during the merge process, because the cache can still link to them. It is necessary to retain a list of such nodes and delete them after the cache is no longer needed.

As this algorithm is based on union-find, the time complexity is pseudolinear if bucketsort can be used or $O(N \log N)$ (N is the edge count, in case of the images also the pixel count) otherwise. However there is one important limitation. The node merge must be done in constant (or potentially logarithmic) time, with a (pseudo)linear (or linearithmic) postprocessing step.

Compared to other connected component tree algorithms based on union-find, this original algorithm guarantees that each path from a tree leaf to the root contains at most one node for each possible value of α. This is important for distance functions such as the one used by [3] where the α is discrete with known upper bound. This reduces the traversal cost to an amortized constant time, independent of the tree size. This speeds up the tree construction at the expense of more complicated tree structure with larger memory footprint.

3.2 C++ Implementation

The common implementation of a component tree is an array of parent indices. This representation is very memory efficient, as the tree requires only $4wh$ bytes of memory. However this representation allows easily only bottom-up tree traversal. Here we used more complex representation for tree nodes.

Every node of the tree consists of a link to a parent node, a set of links to children and a set of leaf pixels. Additionally, the node contains the α value of the edge that created it. The parent pointer of a root node is NULL. The sets of children and pixels allow for very simple iteration over the pixels of a connected component. The set of all pixels of a subtree is a recursive union of pixels of a node and all of its children. The sets are represented by linked lists:

```
struct Node
{
    uint8_t alpha;
    Node * parent;
    Node * first_child, * last_child,
    Node * next_sibling, * prev_sibling;
    Leaf * first_leaf, * last_leaf;
};
```

Children and leaves are represented by doubly linked lists (node removal and list merge have $O(1)$ complexity).

The leaf structure does not contain any data yet. All leaves are stored in an array and the position in the array encodes the pixel position.

```
struct Leaf
{
    Node * parent;
    Leaf * next_sibling, * prev_sibling;
};
```

Compared to an array of parent indices, this representation requires more complicated merging of tree nodes, but the resulting tree is as flat as possible so the tree traversal is shorter. Also, tree balancing is not necessary. Figure 1 shows a sample α-tree and the related linked structure.

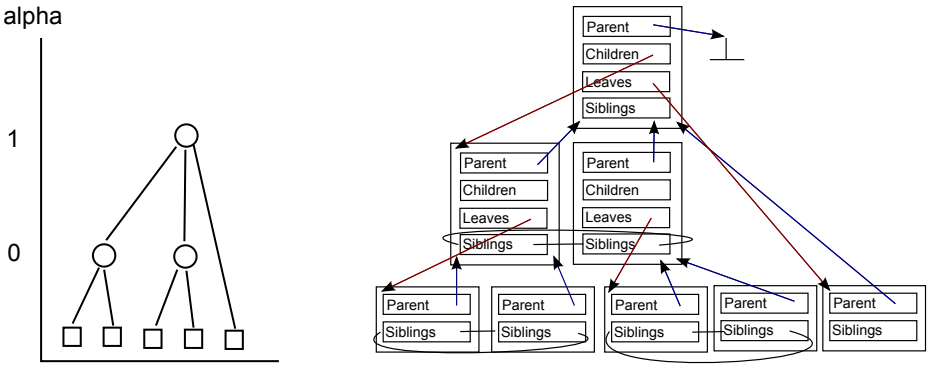

Fig. 1: Sample tree shape and corresponding simplified set of Nodes, Leaves and connecting pointers (for the sake of clarity, only a subset of pointers has been shown).

The maximal height of the tree depends on the datatype of α. In the case of discrete α it is at most the number of α levels that appear in the input image (i.e., at most 256 in case of uint8_t). If α is continuous, it is at most wh when the tree degrades to a simple list. The tree depth of this tree will be always less or equal to the depth of a binary tree implemented by an array of parent links. The cost is lower cache coherence because of dynamic allocation and larger node size.

Two practical strategies for path compression cache are possible:

- Root is cached per node. Cache of nodes visited during tree traversal get updated. This strategy is closer to original Tarjan's paper [14]. However every node needs another pointer that is unused after tree construction. When the node size is not a limiting factor, then this approach has the best results.
- Root is cached per leaf. Only the edge endpoints are updated. Each vertex is usually visited by four edges, so it is a viable strategy, when the node size should be kept as low as possible.

Because merging of two nodes includes transfer (and reconnection) of child nodes and leaf pixels, it has $O(N)$ complexity. Therefore the actual merge must be deferred to the postprocessing step at the end of the algorithm. Since the removal of the nodes must be done in the postprocessing step anyway, it does not introduce new data structure or steps. The node members for the linked list of children are used for a linked list of nodes marked for removal.

3.3 Extension to Multi-threading

Although the previous algorithm is effective, it is inherently single threaded. The only easily parallelizable part is the extraction of edges and their ordering. The tree levels must be processed sequentially. Even in one level, the parallelization would require synchronization of a large portion of the algorithm.

It is however possible to subdivide the input image into multiple parts, build partial trees for them and then merge the resulting trees by a more complicated algorithm that does not need the edges ordered. When the input image is subdivided into regions as square as possible, the number of edges between the subtrees is very low compared to the amount of edges for each subtree. This is the parallelization strategy for other component trees too [2].

The connecting edges are processed sequentially. For both pixels of the connecting edge, the path from the leaf to the tree root is traversed and nodes are found. Contrary to the previous algorithm (where only a subtree root was searched), two tree nodes are required. First node has α less or equal to the connecting edge and the other has α greater than the connecting edge. First nodes are detached and merged in the same way as described by Algorithm 1. Second nodes start two paths to the tree root that need to be merged in a zipper way.

First has α less or equal to the alpha of the connecting edge. These two nodes will be merged in the same way as in the Algorithm 2. The other nodes are the direct parents of the previous nodes, so they have α greater than the connecting edge. These nodes start two paths to tree roots, that will be merged in a way similar to a zipper. This is shown by Algorithm 2.

Algorithm 2 starts by creation of a common root node. This greatly simplifies the zipping process, since the first zip merges the tree roots and subsequent zips merge only inner paths of the tree. The common root is created by merging of roots of T_1 and T_2 when their α is equal or by attaching one to the other. In an unlike case, when all connecting edges have α greater than the roots, a new root node is created and no zipping is done.

4 Experiments

We present here a set of experiments aiming to assert the performance of the proposed computation schemes for building the α-tree. We first present the dataset we are relying on and how we deal with color images, before addressing the respective cases of single and multi-threading. After comparing with [3], this section ends with a discussion about the possibility of processing only partially the tree to lower the computational cost.

4.1 Dataset

We have measured the performance of the proposed algorithms on a few different image datasets. The first is made of the test images of Berkeley segmentation dataset [1] (300 images of size 481×321 pixels). The second dataset has been manually built from 8 wallpaper images of size 1920×1080 pixels (see Figure 2 for a visual composition).

Algorithm 2 Algorithm for merging of two partial α-trees

Require: Trees T_1, T_2; Connecting edges E
Ensure: Merged tree T
  Create common root node.
  for all $e \in E$ **do**
    $p_1, p_2 = \text{points}(e)$
    $n_\{1, 2\} = \text{findNode}(p_\{1, 2\}, \leq \alpha(e))$
    $a = \text{findNode}(p_1, > \alpha(e))$
    $b = \text{findNode}(p_2, > \alpha(e))$
    Detach n_1, n_2 from a, b
    $n = \text{merge}(n_1, n_2)$
    if $\alpha(a) > \alpha(b)$ **then**
      $\text{swap}(a, b)$
    end if
    Attach n to a
    while $a \neq b$ **do**
      if $\alpha(a) = \alpha(b)$ **then**
        $n = \text{merge}(a, b)$
        $a = \text{parent}(a)$
        $b = \text{parent}(b)$
      else
        $a = \text{parent}(a)$
      end if
      if $\alpha(a) > \alpha(b)$ **then**
        Detach n from a
        $\text{swap}(a, b)$
        Attach n to a
      end if
    end while
  end for

Since the segmentation dataset images were too small for a reliable time measurement, we only present here the results obtained with the second dataset. For extra large images, we also used a selection of aerial and satellite photos from mapart.com, namely the 10m SPOT image, Pictometry and DigitalGlobe photos.

Let us mention that images considered here are RGB images, while the definition of the α-tree has been provided only for grayscale images. We have used here the method proposed in [3] to apply the α-tree on RGB images, and we measure the local dissimilarity between neighboring pixels with the Chebyshev distance. Let us consider two colors $\mathbf{c} = (r, g, b)$ and $\mathbf{c'} = (r', g', b')$, the Chebyshev distance between $\mathbf{c}$ and $\mathbf{c}'$ is given by $\max(|r - r'|, |g - g'|, |b - b'|)$. Since the Chebyshev distance is the L_∞ norm of the absolute difference vector $|c - c'|$, the range of possible distance values is kept low and similar to the input range of each color component (i.e. 256 for an 8-bit image). On the opposite, Manhattan or (worst) Euclidean distance lead to a range respectively equal to 3×256 and 256^3 possible distance values. Since this range is to be compared with the various α values, we prefer to set the depth of the α-tree using $A = 256$ rather than $A = 16,777,216$ for practical reasons related to computation time. Let us note,

however, that further options might be explored and that extension of α-tree to color and multivariate images will most probably be a direction of future work.

Fig. 2: A visual composition of the wallpaper dataset showing the 8 wallpaper images used in our experiments.

4.2 Single-threading

We first evaluate the performance of α-tree computation using a single-thread strategy. When the α-tree is built, the average build time is 32ms (15-47ms) for the segmentation dataset and 0.63s (0.44-0.8s) for the wallpapers. According to the program profiling, almost 50% of the build time is spent on bottom-up tree traversal and on updates of the links for path compression. These are the inner loops of the algorithm. All other parts of the algorithm are significantly less time-consuming. The extraction and sorting of edges takes only 10% of the time.

The detailed graphs in Figure 3 shows the detailed timings, number of nodes and rough memory requirements. The construction time increases approximately linearly with the increasing number of the nodes of the tree. The linear dependency is even more significant for larger images as shown on the right graph of this figure. The timing for smaller images is more noisy and contain an outlier probably because of differences in tree structure that turn unimportant with larger input sizes.

4.3 Multi-threading

Performance of parallel tree construction process was measured on a Core i7 2670 CPU with 8 GB of RAM. This processor contains four hyperthreading cores, so it has eight virtual cores. The Figure 4 shows the performance of the parallel construction using multiple threads. The tree construction is mostly memory intensive, so the performance is limited by the memory throughput and not by the raw computation power.

Since the L3 cache and the memory interface is shared between all four cores on Core i7, the memory intensive tree traversal could cause the scaling to stop at a different value than the number of physical or virtual cores. We suspect, that the performance drop for five and more threads is caused by saturation of the L2 cache throughput.

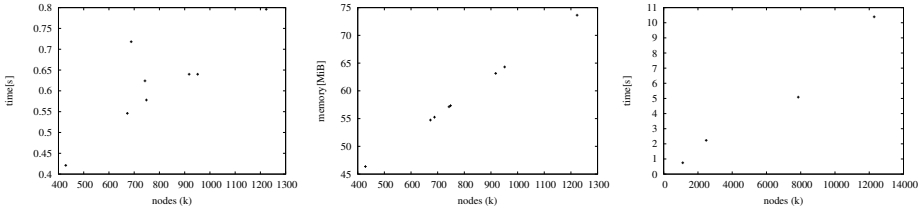

Fig. 3: Dependency of tree construction time [s] (left) and the memory consumption [MiB] (middle) on the number of nodes of the resulting tree, and of tree construction time [s] for large sized images (right).

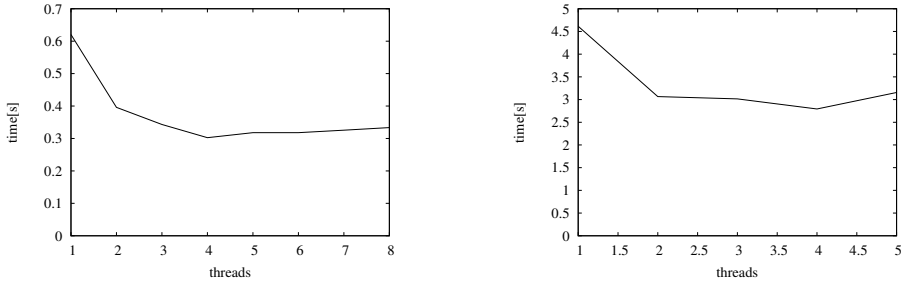

Fig. 4: The average tree build time for increasing thread number. Left: Wallpaper dataset. Right: Satellite+Aerial dataset.

4.4 Comparison with Related Work

In order to assert the efficiency of our algorithm, we compare it with the only available implementation, i.e. [3] which does not rely on the union-find strategy. While our approach requires to collect edges and compress paths, the solution proposed in [3] relies on an insertion process with late merging.

For the sake of comparison, both algorithms have been implemented in Java with the same data structure and compared using two different hardware environments, respectively based on a standard laptop and on a high-performance cluster. This processing environment raises some artifacts (and in particular the influence of the Java garbage collector process) which have been lowered when relevant by computing average times from 10 executions per image (using a trimmed mean to avoid outliers). The processing times given in Table 1 measure the time required to compute the α-tree on the Berkeley Segmentation Dataset (the average time over 300 images is provided), a HD wallpaper image (1920×1080 pixels), and a larger satellite image (> 20 Mpixels). In this context, the focus has to be put rather on the relative times of one algorithm w.r.t. the others than on the absolute time measurement.

From the results given in Table 1, we can see that, while [3] (line 2) improves computation time over the standard union-find strategy (line 1) especially with large images. In order to achieve a fairer comparison, we have developed several extensions of the method proposed [3]. First, we have combined its advantages with the ones offered by

the union-find strategy, leading to better results (line 3) on very large images. Second, we have designed a multithreaded version of [3] to be compared with the multithreaded version of our algorithm presented in this paper. We can see that this multithreaded extension of [3] (line 4) naturally better exploits the availability of a parallel machine w.r.t its singlethreaded counterpart. Finally, our method achieves the best results on small images for its singlethreaded version (line 5) which is penalized by the garbage collector process on large images (not observed in the C++ version), and performs best on all images and environments for its multithreaded version (line 6).

Let us observe however that sometimes the algorithm from [3] and its extensions may lead to relatively close results to ours. This comes from the fact that these two approaches are complementary since their optimizations occur at different steps. This will probably motivate for designing a joint algorithm to achieve further optimization.

Algorithm	(1)	(2)	(3)	(4)
Standard implementation with union-find strategy	560	459	158 004	—
Standard implementation of [3]	371	546	7 801	154 595
[3] using union-find strategy	398	556	8 173	77 256
Multithreaded extension of [3]	301	305	4 555	130 286
Proposed singlethreaded version	351	369	11 297	1 098 163
Proposed multithreaded version	**229**	**183**	**3 327**	**76 866**

Table 1: CPU time comparison with [3] for the different steps of α-tree computation (in milliseconds): (1) average on 300 Berkeley Images (154 401 pixels) with Intel Core Duo T9600 @ 2.80GHz; (2) average on 300 Berkeley Images (154 401 pixels) with 12 Intel Xeon L5640 @ 2.27GHz; (3) Wallpaper HD (2 073 600 pixels) with 12 Intel Xeon L5640 @ 2.27GHz; (4) satellite image (23 403 033 pixels) with 12 Intel Xeon L5640 @ 2.27GHz.

5 Conclusion

Among hierarchical image representations proposed in the scope of mathematical morphology, the recent α-tree model [6] has already shown great interest in image analysis, e.g. in remote sensing or in video processing. In order for this model to be used in real-time or near real-time context, some efficient computing schemes are required.

In this paper, we address this problem and propose several strategies to achieve an efficient computation of the α-tree model. We compare between single-threaded and multi-threaded architectures and show how the α-tree can be built in a reasonable time. Some preliminary experimental results are provided to illustrate the benefits from our algorithms over existing implementations [3].

The main difference of the method presented here from the other connected component algorithms is a more complex tree structure that allows to construct the tree with minimal depth. For certain distance functions, this significantly lowers the traversal time during the construction. The cost is larger memory consumption of the tree.

Future work will deal with the introduction of more global constraints over the tree, namely the ω dissimilarity or other elements of constrained connectivity [11]. Moreover, we would like to build upon these first results in order to offer a wide range of efficient algorithms for computing tree representations, from the max (or min) tree to more recent hyperconnected trees [7]. We believe this is necessary to disseminate these models within the image processing and computer vision communities.

References

1. Martin, D., Fowlkes, C., Tal, D., Malik, J.: A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In: Proceedings of the 8th International Conference on Computer Vision. vol. 2, pp. 416–425. Vancouver, Canada (July 2001)
2. Matas, P., Dokladalova, E., Akil, M., Georgiev, V., Poupa, M.: Parallel hardware implementation of connected component tree computation. In: Field Programmable Logic and Applications (FPL), 2010 International Conference on. pp. 64 –69 (31 2010-sept 2 2010)
3. Merciol, F., Lefèvre, S.: Fast image and video segmentation based on α-tree multiscale representation. In: International Conference on Signal Image Technology Internet Systems. Naples, Italy (November 2012)
4. Najman, L., Couprie, M.: Building the component tree in quasi-linear time. IEEE Transactions on Image Processing 15(11), 3531–3539 (2006)
5. Ouzounis, G., Syrris, V., Gueguen, L., Soille, P.: The switchboard platform for interactive image information mining. In: Soille, P., Iapaolo, M., Marchetti, P., Datcu, M. (eds.) Proc. of 8th Conference on Image Information Mining. pp. 26–30. ESA-EUSC-JRC (October 2012)
6. Ouzounis, G.K., Soille, P.: Pattern spectra from partition pyramids and hierarchies. In: International Symposium on Mathematical Morphology. pp. 108–119. Verbania-Intra, Italy (2011)
7. Perret, B., Lefèvre, S., Collet, C., Slezak, E.: Hyperconnections and hierarchical representations for grayscale and multiband image processing. IEEE Transactions on Image Processing 21(1), 14–27 (January 2012)
8. Salembier, P., Garrido, L.: Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval. IEEE Transactions on Image Processing 9(4), 561–576 (2000)
9. Salembier, P., Oliveras, A., Garrido, L.: Anti-extensive connected operators for image and sequence processing. IEEE Transactions on Image Processing 7(4), 555–570 (1998)
10. Serra, J.: Anamorphoses and function lattices. In: Dougherty, E.R. (ed.) Mathematical Morphology in Image Processing", chap. 13, pp. 483–523. Marcel Dekker, New York (1993)
11. Soille, P.: Constrained connectivity for hierarchical image partitioning and simplification. IEEE Transactions on Pattern Analysis and Machine Intelligence 30(7), 1132–1145 (July 2008)
12. Soille, P., Grazzini, J.: Constrained connectivity and transition regions. In: International Symposium on Mathematical Morphology. pp. 59–69. Groningen, The Netherlands (August 2009)
13. Soille, P., Najman, L.: On morphological hierarchical representations for image processing and spatial data clustering. Lecture Notes in Computer Science 7346, 43–67 (2012)
14. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM 22, 215–225 (1975)